Quantum Computing (CST Part II)

Lecture 12: Quantum Complexity

By any objective standard, the theory of computational complexity ranks as one of the greatest intellectual achievements of humankind.
Scott Aaronson

Resources for this lecture

Nielsen and Chuang chapter 3 gives an overview of theoretical computer science, including computational complexity.

Automata and computability, (D. C. Kozen), is probably the best book for a comprehensive introduction to automata, but this is not required (or even recommended) reading for this course (this book is pointed out purely for reference).

Recap: Church-Turing

The Church-Turing thesis states:

> *A function on the natural numbers can be calculated by an effective method if and only if it is computable by a Turing machine.*

We know that quantum computing does not violate the Church-Turing thesis, which concerns computability, however we have reason to suspect that it may violate the Strong Church-Turing thesis:

> *Any algorithmic process can be simulated efficiently using a probabilistic Turing machine,*

which concerns complexity (here "efficiently" is taken to mean, with only a polynomial time overhead).

Quantum complexity: the big picture

- We have already seen that quantum mechanics enables information processing tasks that cannot be achieved classically, for example superdense coding, and quantum key distribution.
- Furthermore, we have seen that quantum mechanics definitely allows some computational tasks to be achieved more quickly than in the classical case, for example Grover search...
- ...and in some cases, this speed-up is apparently exponential, for example Shor's algorithm.
- Indeed, the nature of entangled spaces seems to be such that quantum mechanical systems fundamentally have exponentially more computational power than classical systems do.

All this begs the question of whether there is some fundamental complexity class separation between tasks that can be achieved efficiently (in polynomial time) quantumly and those that can be classically.

Finite automata

A finite automata consists of:

- A set of n_s states.
- An input alphabet of size n_a.
- A set of state transitions: usually represented in the form of a $n_s \times n_s$ matrix for each of the n_a letters.
- An initial "start" state.
- An "accept" state (marked by a black circle).

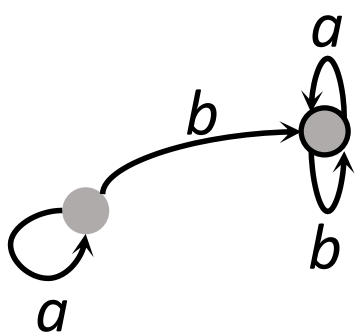
The *accepted language* is the set of strings of letters from the alphabet, such that the final state is the "accept" state.

Note that in general there could be multiple accept states (but in this lecture we only deal with examples with a single accept state).

Deterministic finite automata

Deterministic automata have the property that for each state-letter pair, there is only one outgoing arrow. The transition matrices are such that, if $|i\rangle$ is the start state, and $s_1 s_2 \cdots s_n$ is input the string, then the final state is $|f\rangle = M_{s_n} M_{s_{n-1}} \cdots M_{s_1} |i\rangle$ (where M_{s_i} is the transition matrix for the ith symbol, s_i). It follows that the transition matrices of deterministic finite automata have exactly one 1 in each column.

Example:



With the left-hand state, $|0\rangle$, as the starting state (and $|1\rangle$ is the accept state).

The transition matrices are:

$$M_a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \; ; \; M_b = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$$

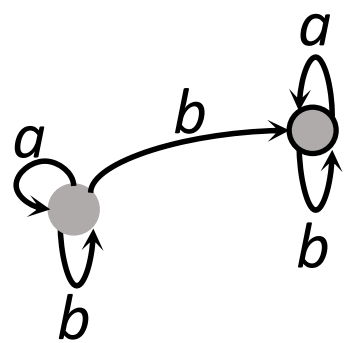
It can be seen that the accepted language is the set of all strings containing at least one "b", e.g., if the string abb is read we get, $M_b M_b M_a$ multiplied by the start state:

$$\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Nondeterministic finite automata

Nondeterministic finite automata can have any number of outgoing arrows for each state-letter pair, so the transition matrices are general binary matrices. A string is part of the accepted language if there is *some* path finishing in the accepted state.

Example:



With the left-hand state, $|0\rangle$, as the starting state (and $|1\rangle$ is the accept state).

In this example, the transition matrices are:

$$M_a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and

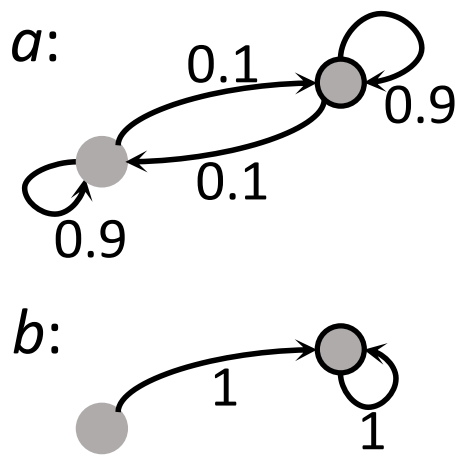
$$M_b = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

And it is the still the case that the accepted language is the set of all strings containing at least one "b".

Probabilistic automata

Probabilistic automata are essentially *Markov chains*, with state transitions being probabilistic, and thus the transition matrices contain fractional values, such that each column sums to one. The accepted language can be defined either as the set of all strings that end in the final state with certainty, or with probability above some threshold.

Example:



With the left-hand state, $|0\rangle$, as the starting state (and $|1\rangle$ is the accept state).

In this example, the transition matrices are:

$$M_a = \begin{bmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{bmatrix}$$

and

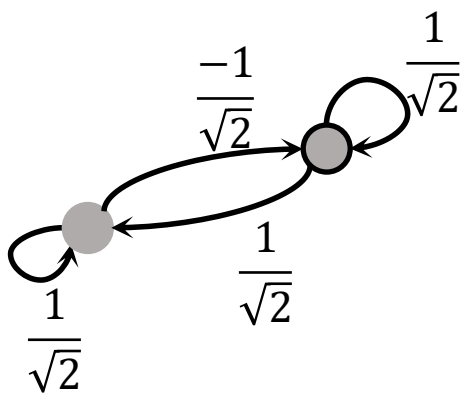
$$M_b = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$$

So all strings (with at least one symbol) are accepted with some probability, and all strings ending with a "b" are accepted with certainty.

Quantum automata

In quantum automata, the transition matrices are unitary matrices consisting of positive and negative complex numbers. A special case of quantum automata are reversible automata, where the transition matrices are binary permutation matrices (exactly one 1 in each column and row).

Example:



With the left-hand state, $|0\rangle$, as the starting state (and $|1\rangle$ is the accept state).

This single-letter alphabet automata has transition matrix:

$$M_a = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$$

We can see that:

$$M_a M_a = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

So, if we start in state $|0\rangle$ there are two paths of length 2 from $|0\rangle$ back to itself ($|0\rangle \to |0\rangle \to |0\rangle$ and $|0\rangle \to |1\rangle \to |0\rangle$), but these interfere is such a way that there is actually zero chance of ending up in $|0\rangle$.

Turing machines

Turing machines are deterministic finite automata, equipped with an infinitely long read-write tape, upon which the input string is initially written. At any time a "head" is over one space on the tape, and can read the symbol written there (initially the head is at the left-hand end of the tape). The action of a Turing machine is thus:

At a given time, the DFA is in a certain state, and the head is over a symbol which it reads. Given this state-symbol pair, a transition function determines:

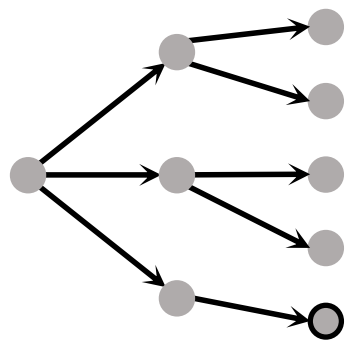
- Which symbol to overwrite on the current space on the tape.
- Whether to move the head left or right.
- Which next state the DFA moves to.

A Turing machine accepts the input if it halts in an accept state.

The Turing machine is a sufficiently general model for computation to capture entirely that which can reasonably be thought of as mathematically computable. The class of problems that can be decided in polynomial-time on a Turing machine is denoted P.

Nondeterministic Turing machines

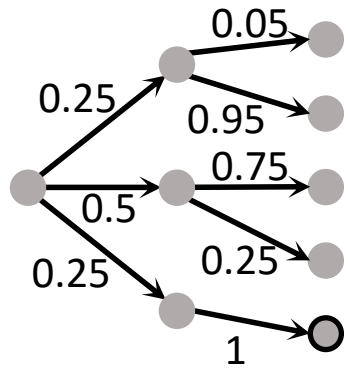
If instead of a single action (overwritten symbol, move left or right and state transition) we allow a set of possible actions, then we get the *nondeterministic Turing machine*. This can be thought of as a tree, where each branching process represents the variety of possible actions at a given time. A string is accepted if there is some path through the tree to an accept state.



If the height of the tree is bounded by a polynomial in the length of the input string, then the language is in NP. Clearly $\mathsf{P} \subseteq \mathsf{NP}$, as each decision point could just consist of one single branch.

Probabilistic Turing machines

Probabilistic Turing machines are similar in appearance to nondeterministic Turing machines, but now the branches represent a probability distribution over possible next actions:



The complexity class BPP is the set of languages, L, for which there is a probabilistic Turing machine, M, running in polynomial time with:

$$P(M \text{ accepts } w) = \begin{cases} > \frac{2}{3} & \text{if } w \in L \\ < \frac{1}{3} & \text{if } w \notin L \end{cases}$$

Note $\frac{2}{3}$ is arbitrary – all that is required is that we have a constant fraction greater than $\frac{1}{2}$ (and similarly one less than $\frac{1}{2}$ for the $\frac{1}{3}$ term). Clearly each probability distribution in a probabilistic Turing machine could consist of a single deterministic branch, so P $\subseteq$ BPP.

Quantum Turing machines

Quantum Turing machines are like probabilistic Turing machines, but now complex amplitudes are associated with each possible next move. It is also necessary that the linear transformation defined by the machine is unitary.

The complexity class BQP is the set of languages, L, for which there is a quantum Turing machine, M, running in polynomial time with:

$$P(M \text{ accepts } w) = \begin{cases} > \frac{2}{3} & \text{if } w \in L \\ < \frac{1}{3} & \text{if } w \notin L \end{cases}$$

It has been shown that the quantum Turing machine generalises the probabilistic Turing machine, so BPP $\subseteq$ BQP.

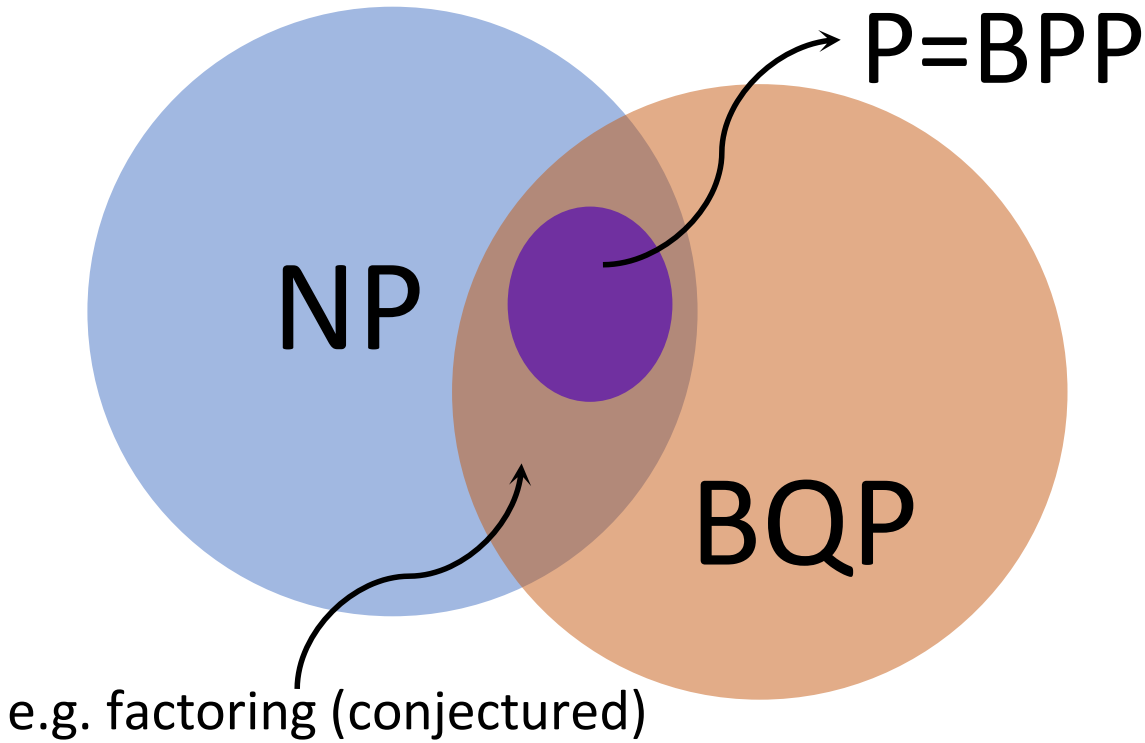
Relationships between complexity classes

Many complexity class inclusions remain open problems in theoretical computer science. We have already seen that $\mathsf{P} \subseteq \mathsf{NP}$ and $\mathsf{P} \subseteq \mathsf{BPP} \subseteq \mathsf{BQP}$, but are these proper subsets?

- $\mathsf{P} = \mathsf{NP}$? is the most famous open problem in theoretical computer science, and the vast majority of theorists believe $\mathsf{P} \neq \mathsf{NP}$.
- We do not know whether BPP is a subset of NP or vice versa, but it is conjectured that $\mathsf{P} = \mathsf{BPP}$.
- As factoring is widely believed to be super-polynomial classically (even with a probabilistic Turing machine), the existence of Shor's algorithm is taken as evidence that $\mathsf{BPP} \neq \mathsf{BQP}$.
- It is widely believed that NP-complete problems cannot be solved in polynomial time on a quantum computer (unless $\mathsf{P} = \mathsf{NP}$), so $\mathsf{NP} \not\subseteq \mathsf{BQP}$.
- ...but it is also believed that there are problems outside of NP which *can* be solved in polynomial time on a quantum computer, so $\mathsf{BQP} \not\subseteq \mathsf{NP}$.

Complexity class inclusions

A Venn diagram of complexity class inclusions as we conjecture them:



BPP $\subsetneq$ BQP? – A purely theoretical question?

Theoretical computer science has driven the development of the theory of computational complexity (in particular complexity classes), so it is worth briefly addressing the practical significance of this. To do so, we'll look at a few pertinent questions one may have:

- Why is super-polynomial so bad?

  Exponentially complex functions really are intractable in practise. In the near-term it is expected that quantum simulations will be undertaken on quantum computers that cannot be achieved classically. Moreover, if and when we have full-scale quantum computers (say of the order of 1000s of error-corrected qubits), they will be able to factor numbers which could not conceivably be factored on classical computers (using currently known techniques).

BPP $\subsetneq$ BQP? – A purely theoretical question? (cont.)

- Are speed-ups within the polynomial class (for example the quadratic speed-up achieved by Grover search) worthwhile in practise?

  On this there is some discrepancy amongst experts: some think that, given the continual improvements in classical computers, exponential speed-ups will be necessary for quantum computing to become a viable technology; others think that, because Moore's law will eventually break down, even more modest polynomial-class speed-ups will make quantum computing an attractive post-CMOS technology.

- Does it matter that we don't know whether BPP $\subsetneq$ BQP?

  From a practical point of view, not really. If we do not *know* that the class of problems that can be efficiently solved using quantum computers is greater than that for classical computers, but we do *know* some examples of polynomial-time quantum algorithms for which there are only super-polynomial classical counterparts (for example, factoring) then it is still reasonable, from a technological point of view, to build quantum computers.

Summary

In this lecture we have covered:

- The bigger picture of quantum computability and complexity.
- Various finite automata, including quantum automata.
- Turing machines and complexity classes.
- The practical relevance if BPP $\subsetneq$ BQP.